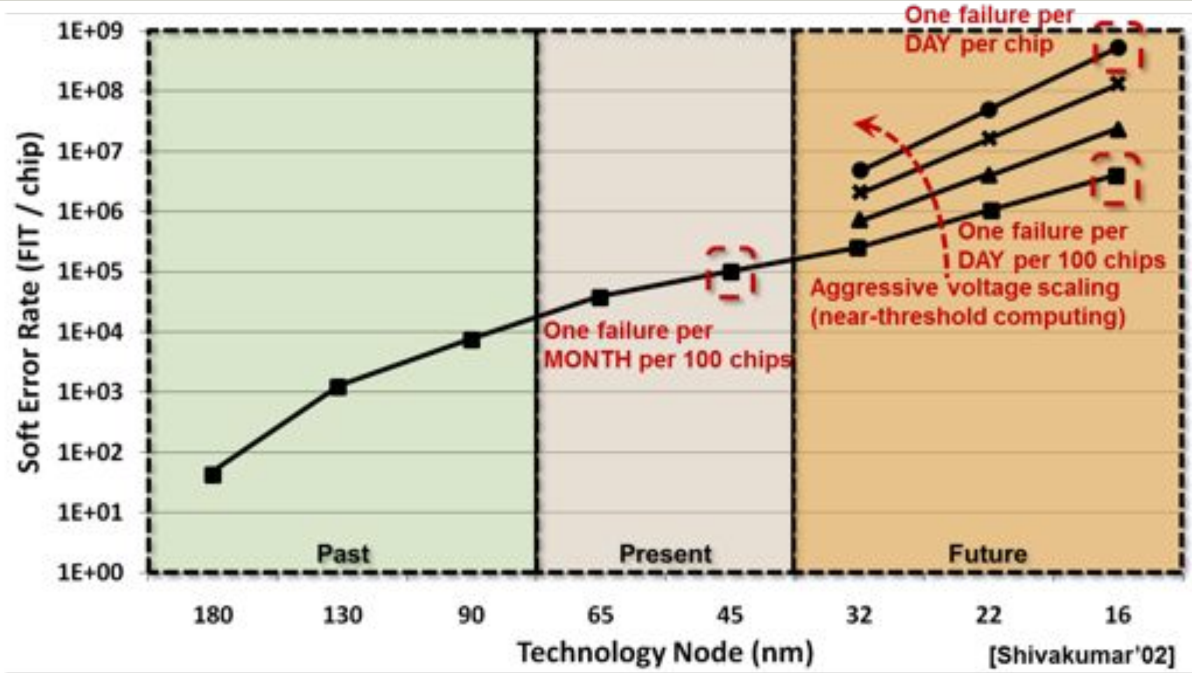
SDCTune: A Model for Predicting the SDC Proneness of an Application for Configurable Protection

Qining Lu, **Karthik Pattabiraman**
University of British Columbia (UBC)

Jude Rivers, Meeta Gupta
IBM Research T.J. Watson

Motivation: Transient Errors



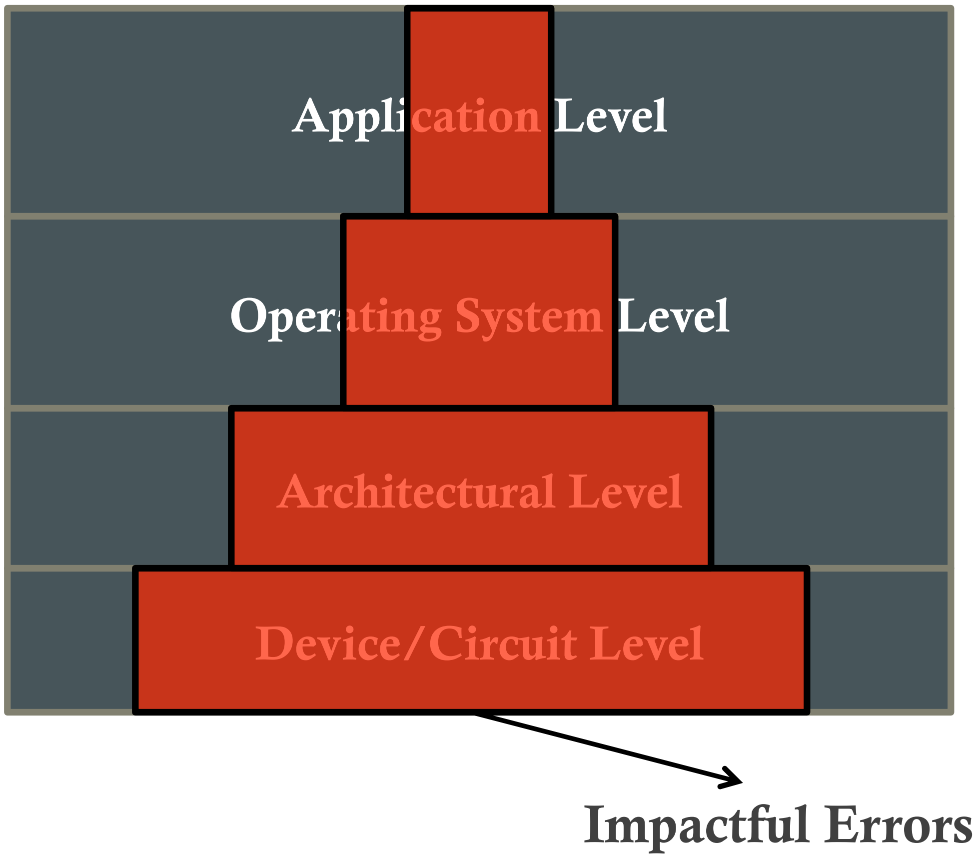
Particle strikes, temperature, etc.,

Transient hardware faults

Source: Feng et. al., ASPLOS'2010

Transient hardware errors (aka. Soft errors) ***increase*** as **feature sizes** ***shrink***

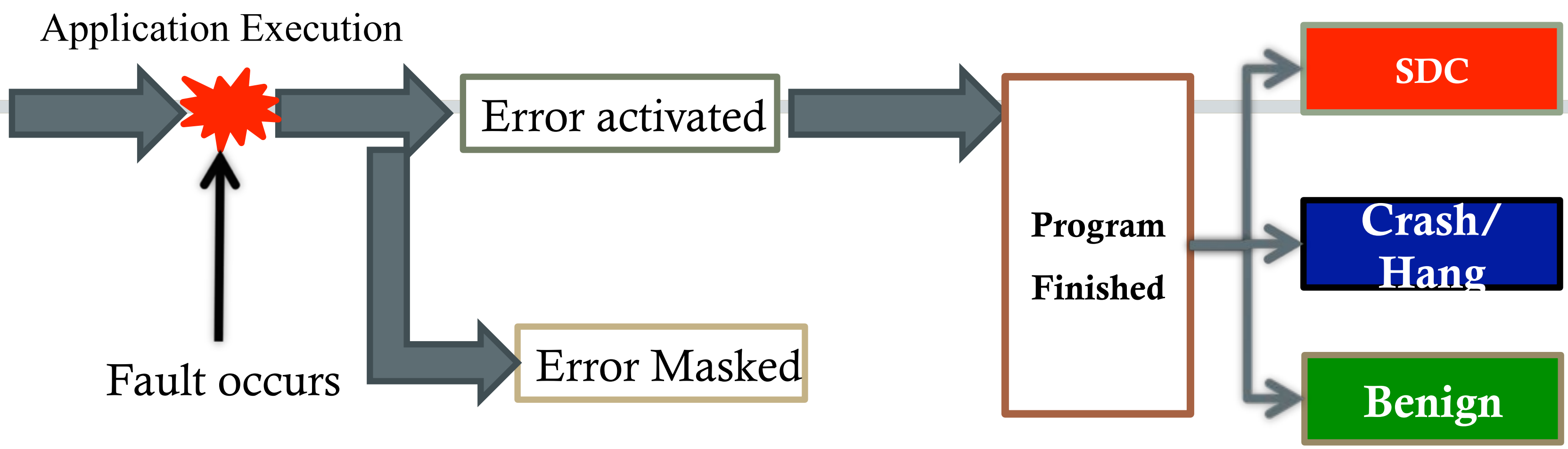
Motivation: Application-level Techniques



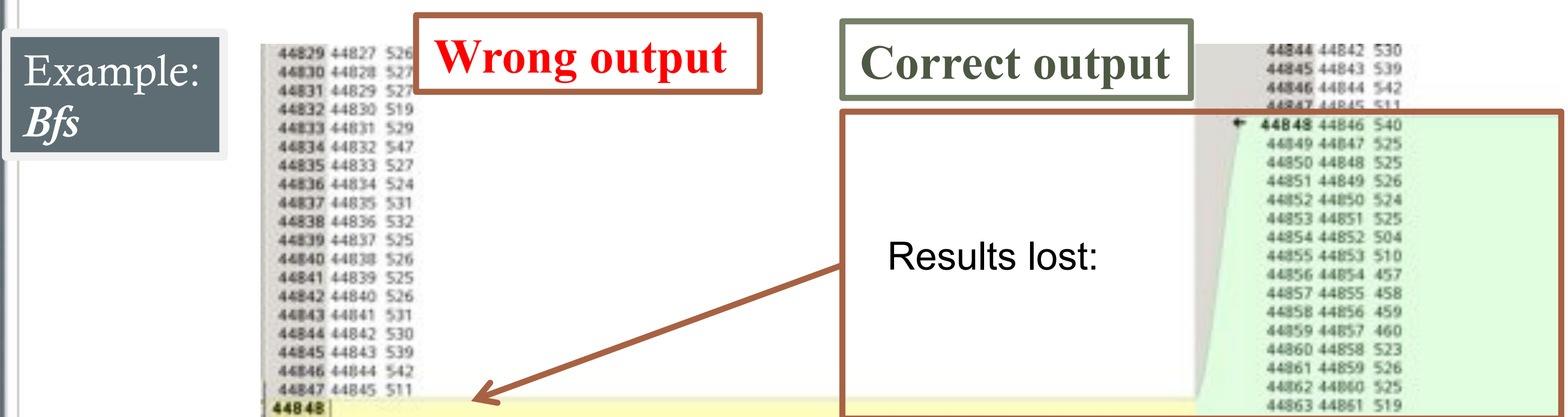
Only a fraction of the errors at the ***circuit level*** *impacts the* ***application***

More ***economical*** to deploy techniques at **application**

Motivation: Silent Data Corruption (SDC)



Silent Data Corruption (SDC): Our focus in this paper

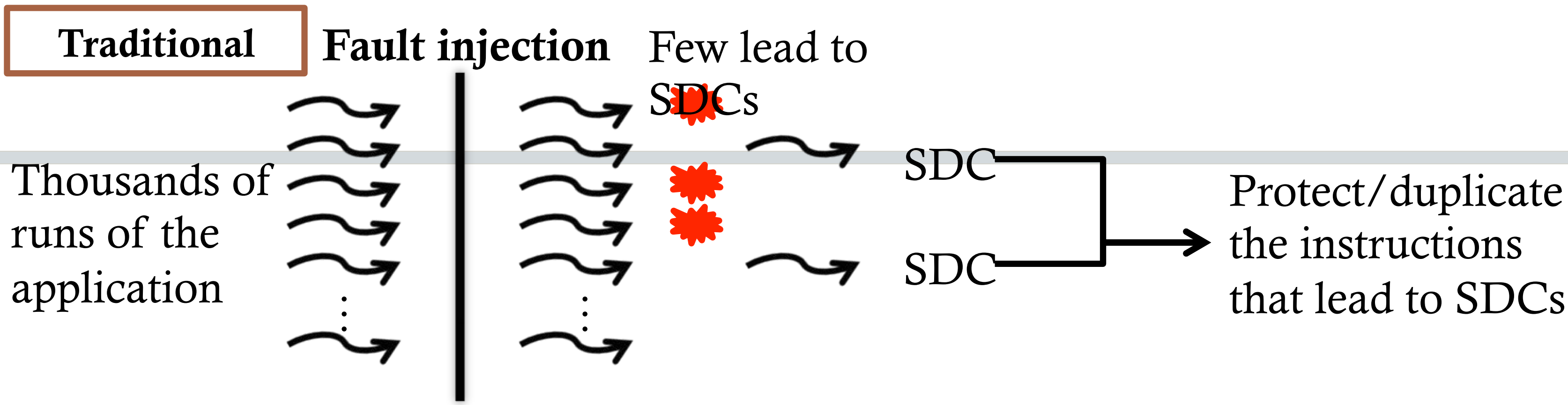


Our Goals

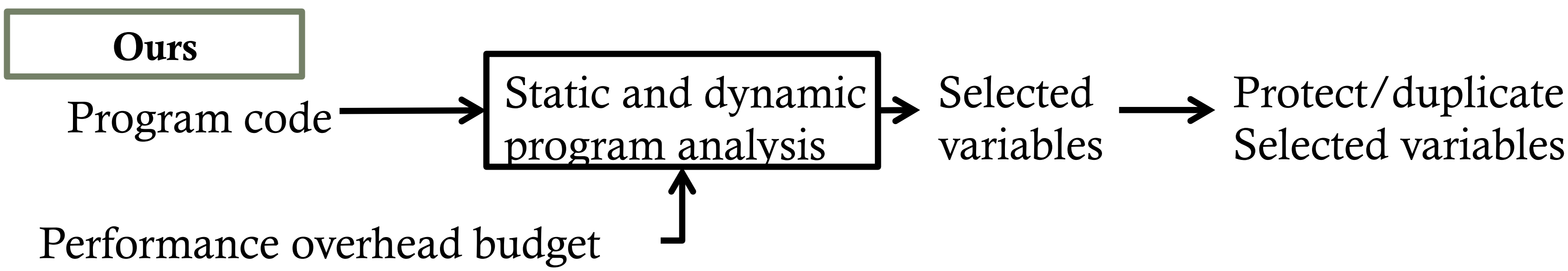
- Detect **Silent Data Corruption (SDC)**
- **High** Coverage **with Low** Overhead
- *Configurable* **protection overhead**

Selectively protect highly SDC-prone variables in program

Traditional approaches Vs. Our approach



- Time consuming (runs application thousands of times)
- Need to manually choose variables to protect



- Time saving (dynamic analysis only runs the application once)
- Automatically choose variables to protect subject to performance

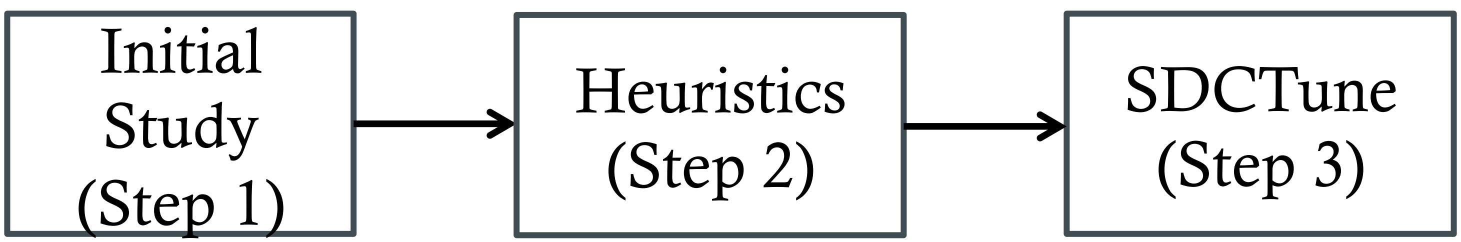
Fault model

- Single bit flip fault
- One fault per run
- Errors in registers and execution units
- Program data that is visible at architectural level

- Motivation and Goal
- Approach
- Evaluation and Results
- Conclusion

Overall Approach

- Step 1: Perform fault injections to understand SDC characteristics of code constructs
- Step 2: Heuristics identifying code regions prone to SDC causing faults
- Step 3: SDCTune model building and protection



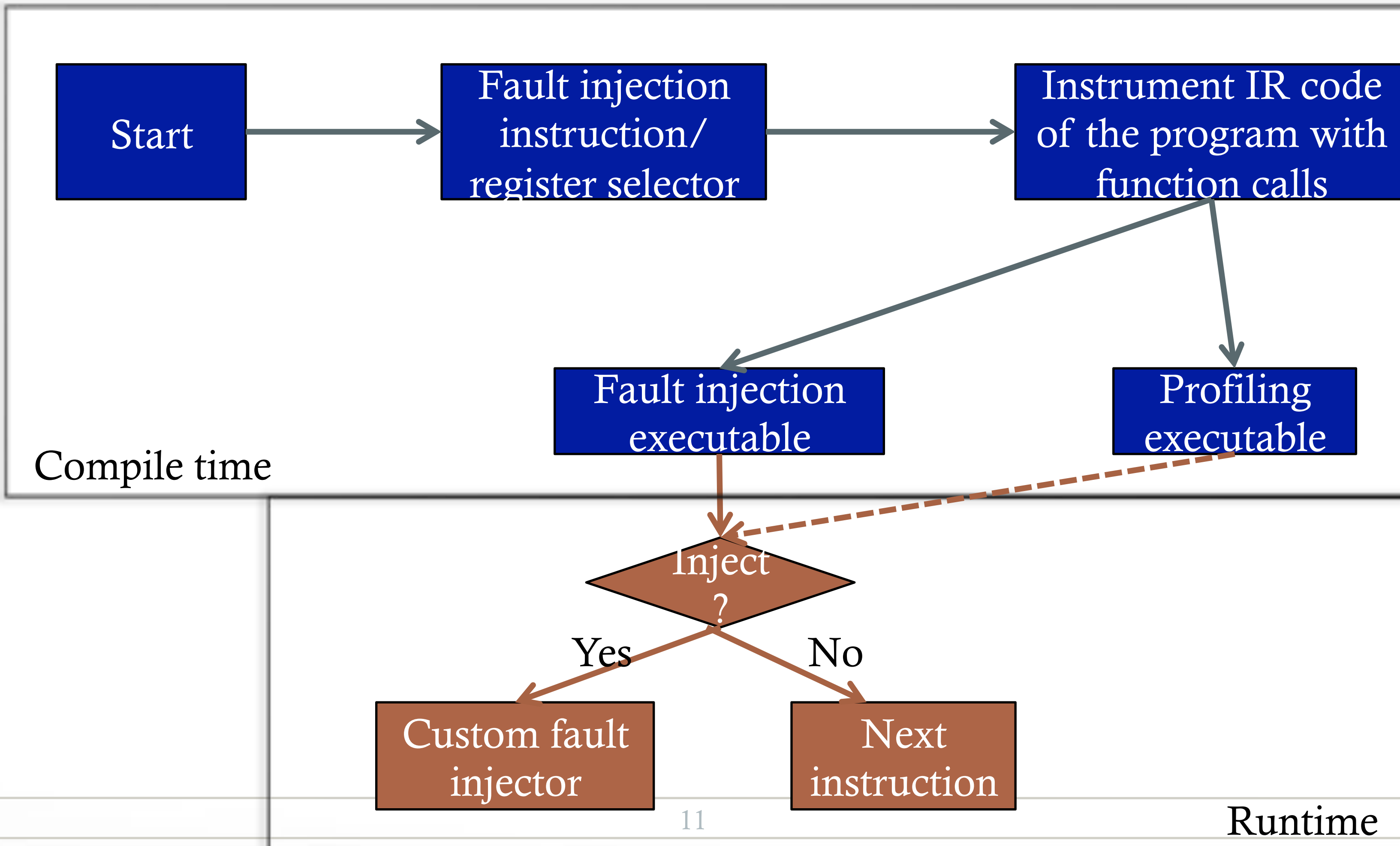
Initial study: Goals

- **Initial fault injection experiments**
  - *The goal is to understand the reasons for SDC failures*
  - *Used to formulate heuristics for selective protection*

- **Manually inspect why SDC occurs**
  - *Highly executed instructions cover most SDCs*
  - *Not all highly executed instructions should be protected*
  - *Find common patterns used for developing heuristics*

Initial Study: Method

- Performed using LLFI, high level fault injector validated for SDC-causing errors [DSN'14]



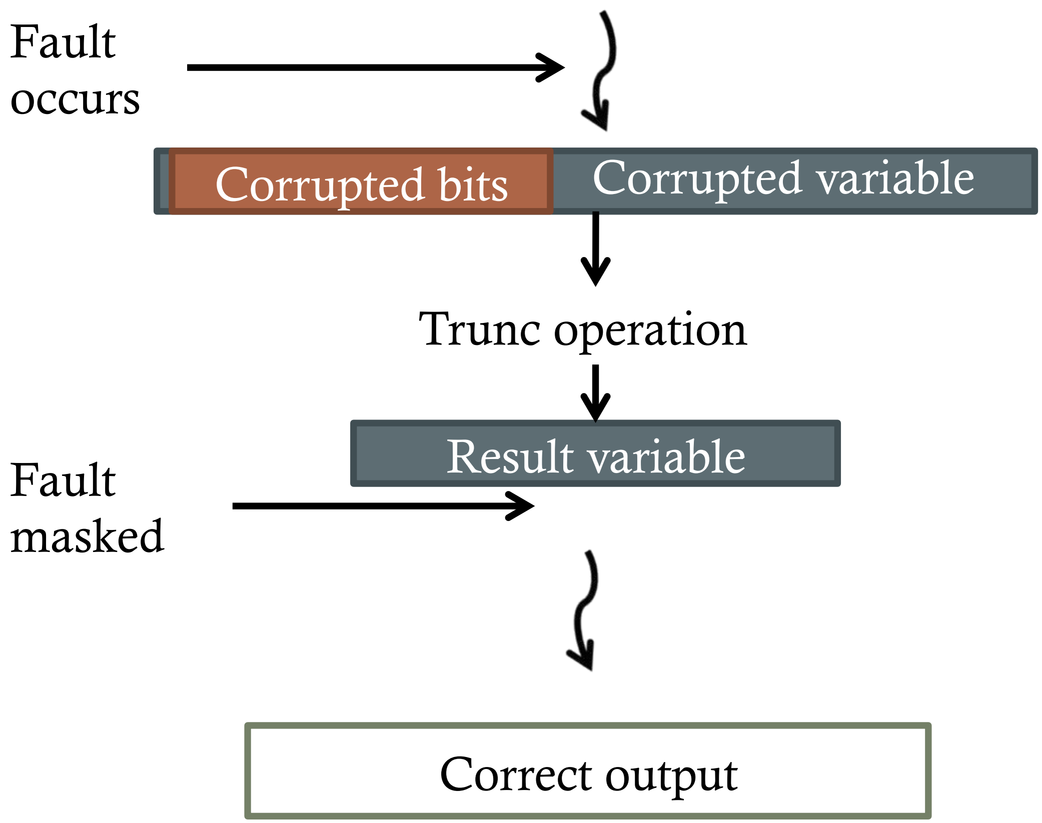
Initial study: Findings

- **SDC proneness of instruction depends on*:***
  - ***The fault propagation in its data dependency chain***
  - ***The SDC proneness of the end point of that chain***
- **End points of data dependency chain*:***
  - ***Store operations***
  - ***Comparison operations***

Need heuristics for fault propagation, store operations, comparison operations

Heuristics: Fault propagation

HP1: The SDC proneness of an instruction will decrease if its result is used in either fault masking or crash prone instructions



Heuristics: Store operations

Category	Description	Major related features
Addr NoCmp	The stored value is used in calculating memory addresses but not comparison results	Data width
Addr Cmp	The stored value is used in calculating both memory addresses and comparison results	Data width and control flow deviation
Cmp NoAddr	The stored value is used in calculating comparison results but not memory addresses	Resilient or Unresilient comparison
NoCmp NoAddr	The stored value is neither used in memory address calculation nor comparison results	Used in output or not

HS1: Addr NoCmp stored values have low SDC proneness in general

HS2: Addr Cmp stored values have higher SDC proneness than Addr NoCmp

<More heuristics in paper>

Heuristics: Comparison operations

HC1: Nested loop depths affect the SDC proneness of loops' comparison operations.

```
void BZ2_hbMakeCodeLengths
      (...){
  while(nHeap>1){ //outer loop
    ...
    while(weight[tmp]<weight[
          heap[zz>>1]]){
      // inner loop
      Heap[zz]=heap[zz>>1];
      zz>>1;
    }
  }
}
```

SDC proneness of "***nHeap>1***" higher than "***weight[tmp]<weight[heap[zz>>1]]***"

<More heuristics in paper>

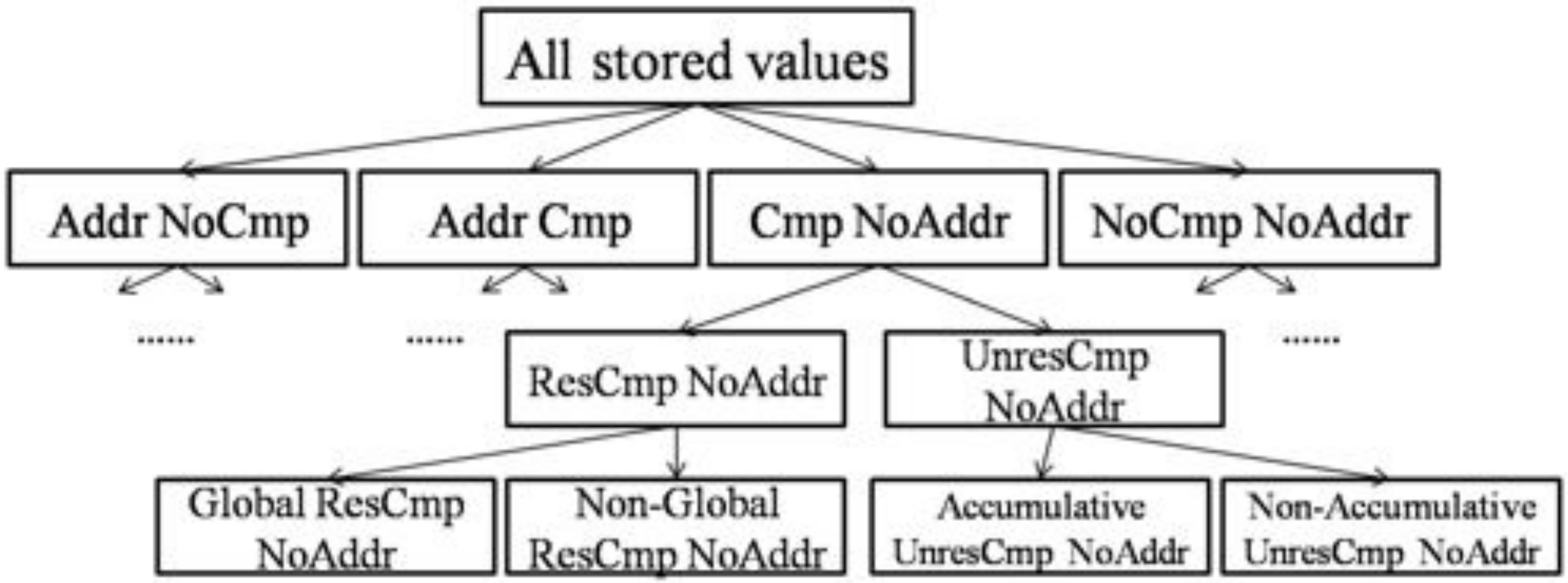
SDCTune: Build model

- **Classification**
  - ***Different types of usage are usually independent of each other***
  - ***Classify the stored values and comparison values according to the heuristic features we observed before***
- **Regression**
  - ***With same type of usages, SDC rate may show gradually correlations to several features***

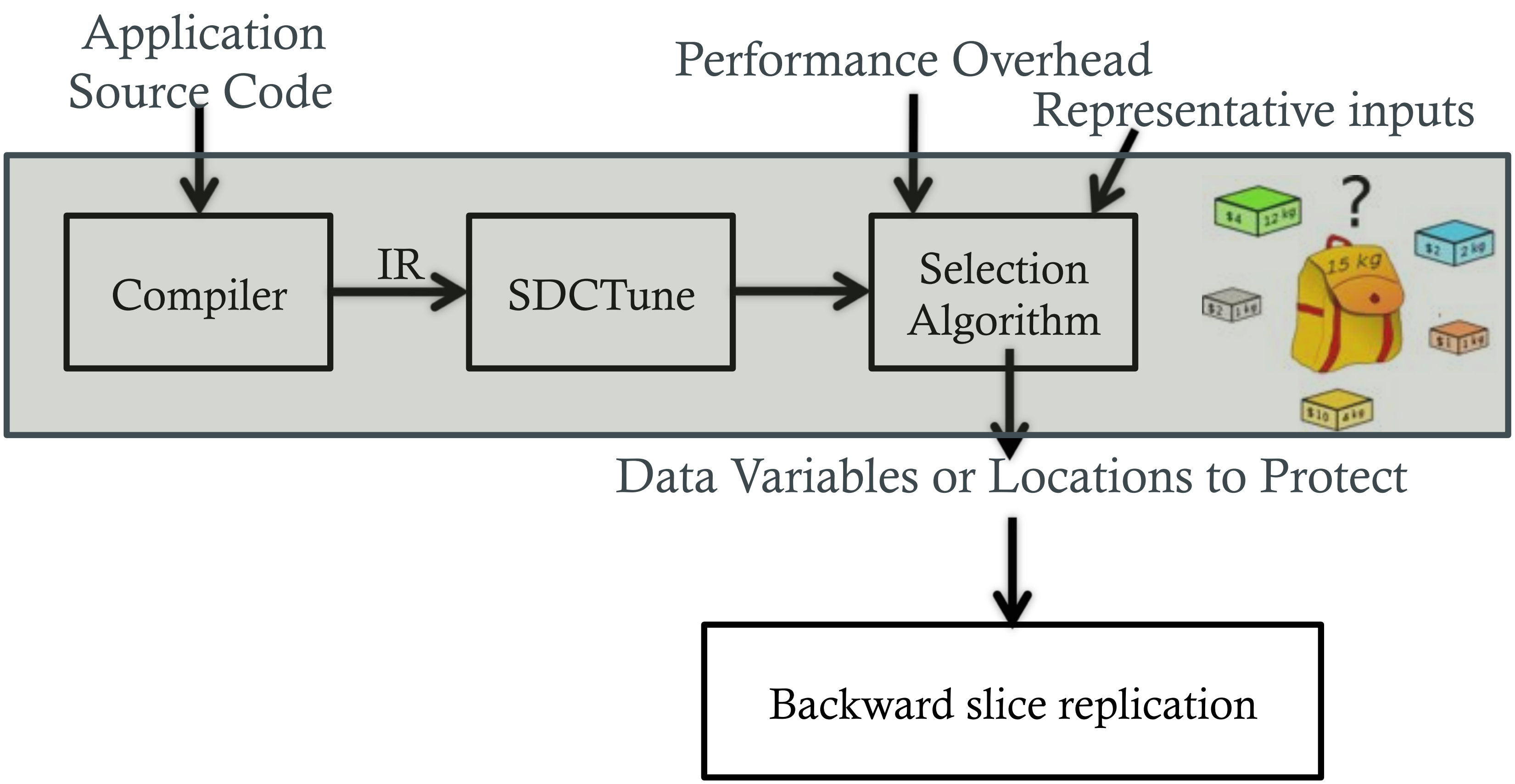
52 features in total used in the model

SDCTune: Example model

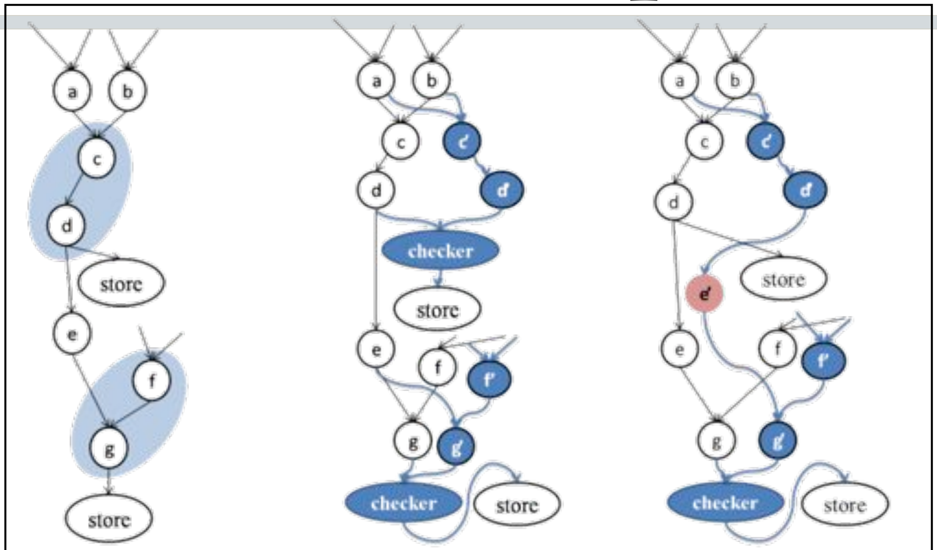
Example: *tree structure for Store*



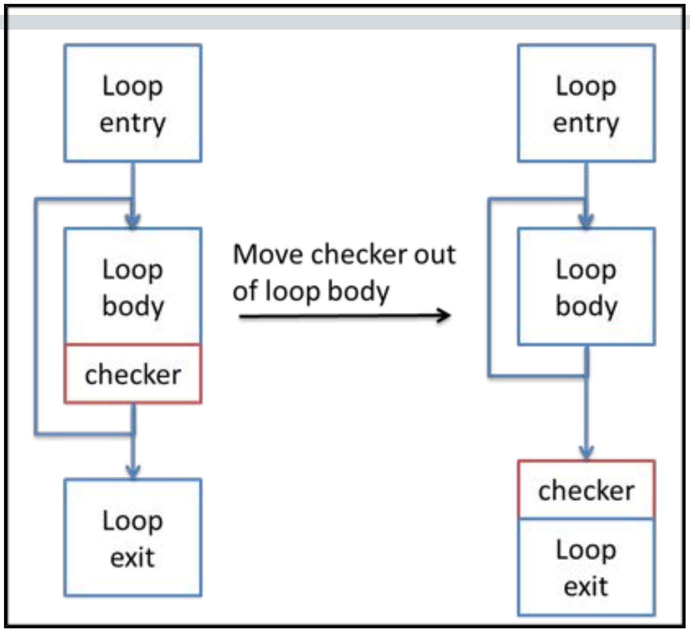
SDCTune: Selection Algorithm



SDCTune: Optimizations



(a) Data dependency of detector-free code (b) Basic detector instrumented (c) *concatenate* duplicated instructions

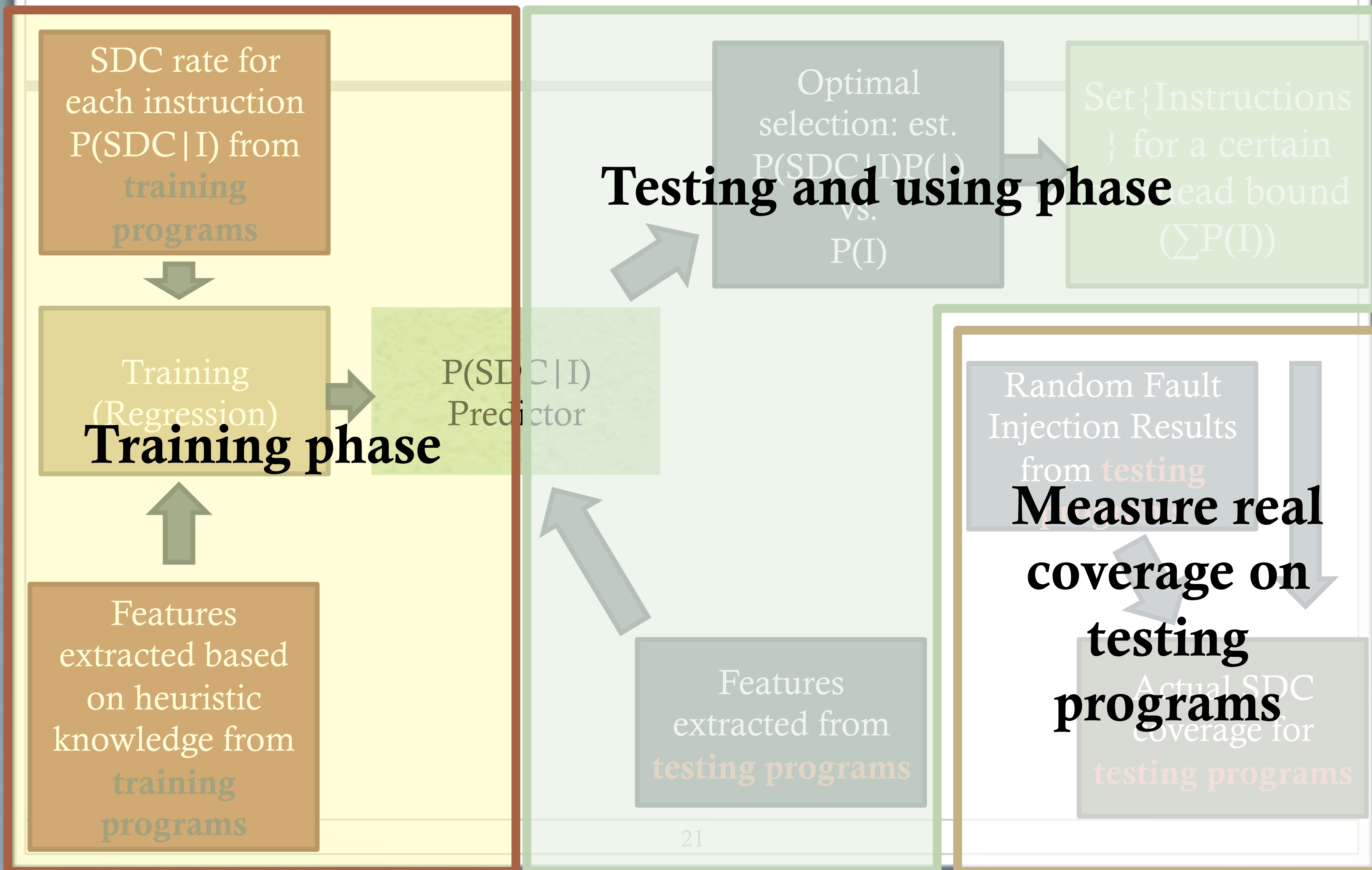


Adding the instructions to the protection set to save checkers

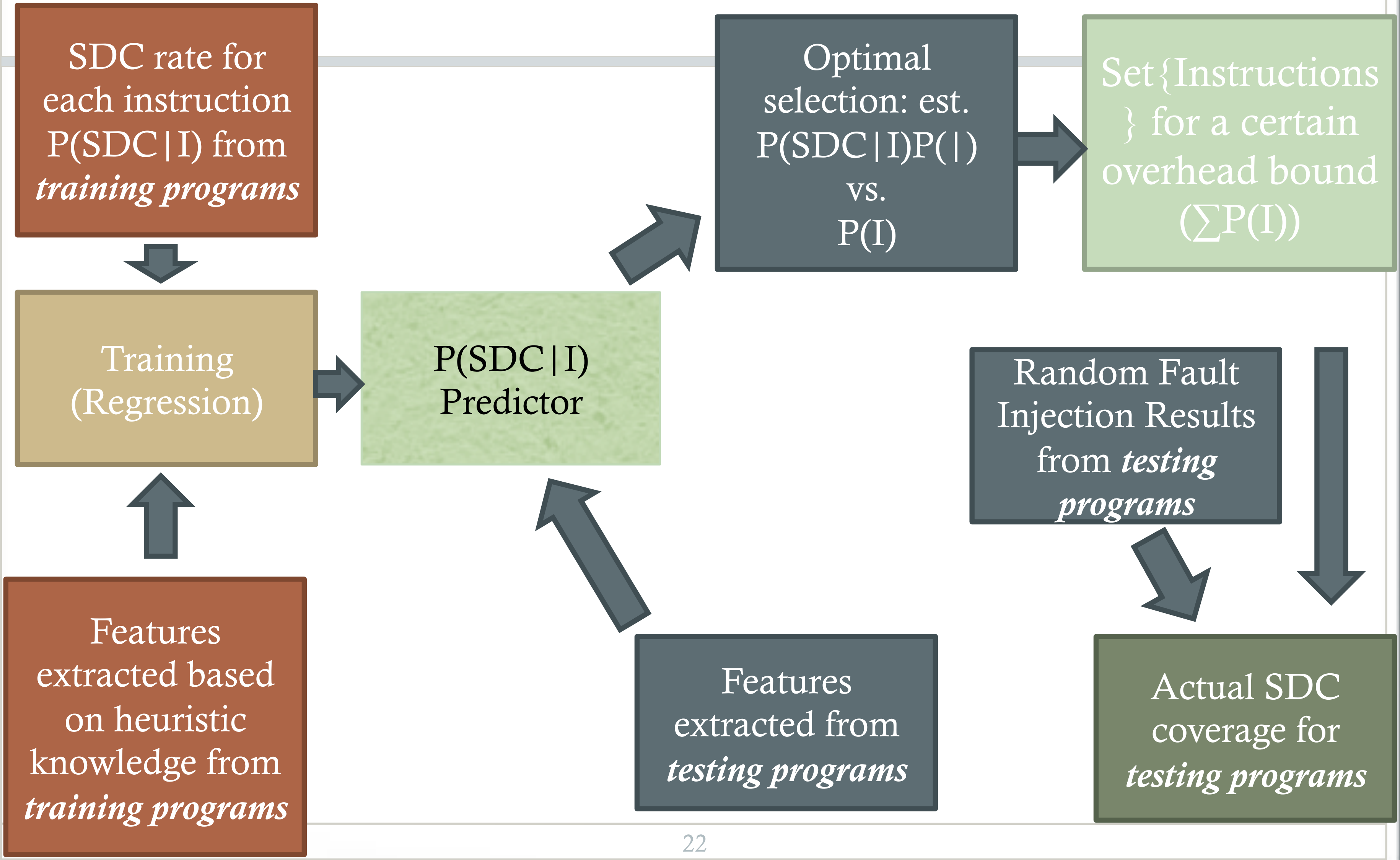
Move checker out of loop body

- Motivation and Goal
- Approach
- Evaluation and Results
- Conclusion

Evaluation: Work Flow



Evaluation: Work Flow



Evaluation: Benchmarks

Training programs

Program	Description	Benchmark suite
IS	Integer sorting	NAS
LU	Linear algebra	SPLASH2
Bzip2	Compression	SPEC
Swaptions	Price portfolio of swaptions	PARSEC
Water	Molecular dynamics	SPLASH2
CG	Conjugate gradient	NAS

Testing programs

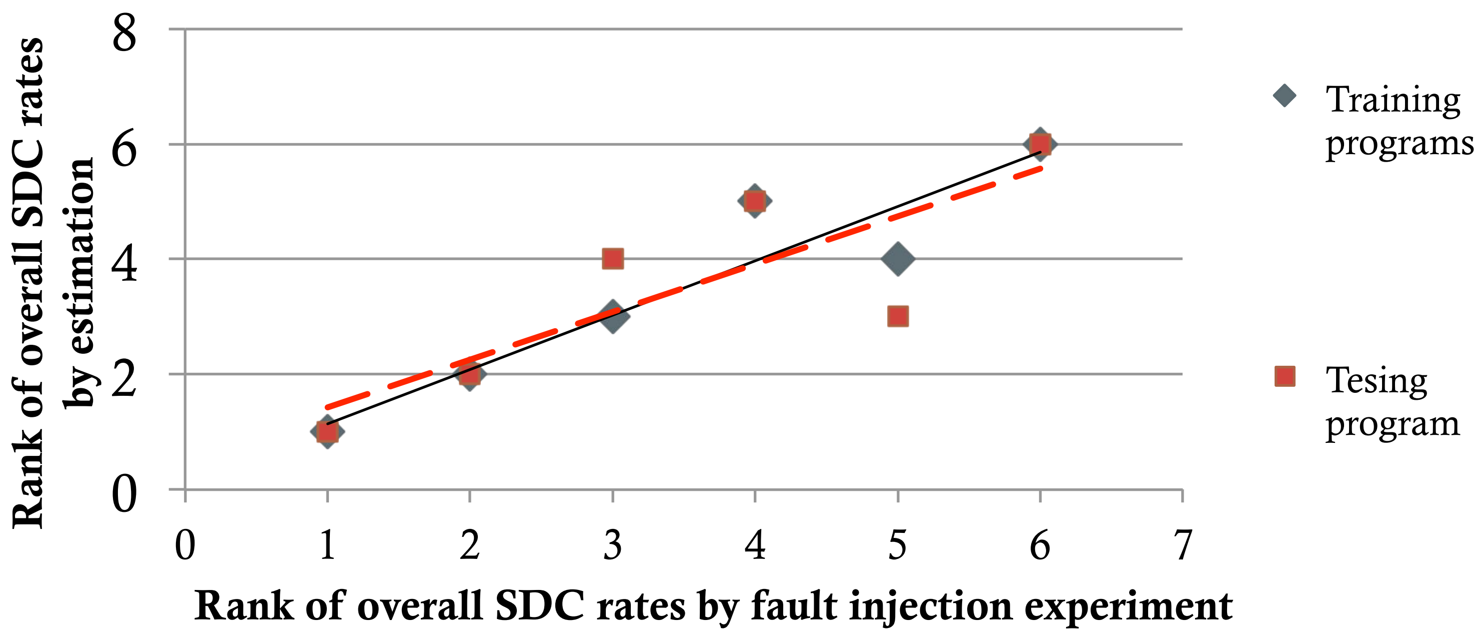
Program	Description	Benchmark suite
Lbm	Fluid dynamics	Parboil
Gzip	Compression	SPEC
Ocean	Large-scale ocean movements	SPLASH2
Bfs	Breadth-First search	Parboil
Mcf	Combinatorial optimization	SPEC
Libquantum	Quantum computing	SPEC

Evaluation: Experiments

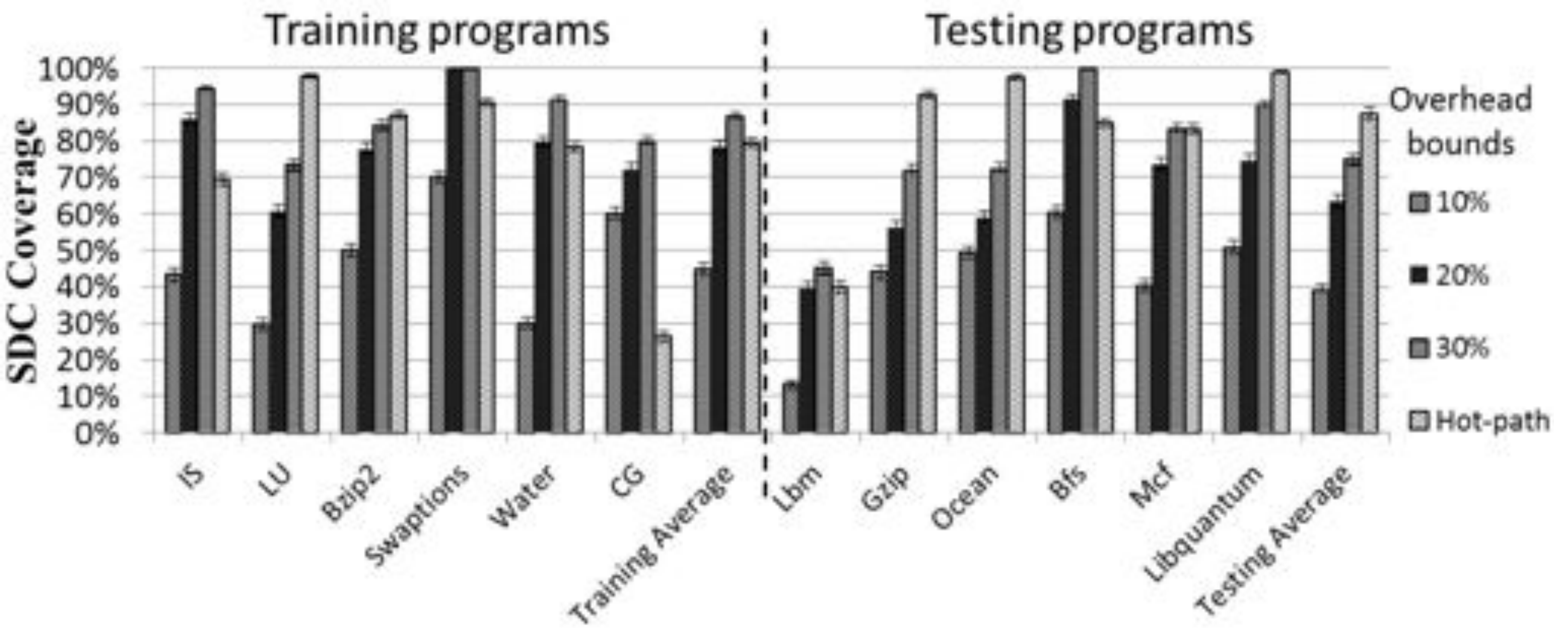
- **Estimate overall SDC rates using SDCTune and compare with fault injection experiments**
  - Measure correlation between predicted and actual
- **Measure SDC Coverage of detectors inserted using SDCTune for different overhead bounds**
  - Consider 10, 20 and 30% performance overheads
- **Compared performance overhead and efficiency with full duplication and hot-path duplication**
  - Efficiency = SDC coverage / Performance overhead

Results: Overall SDC Rates

	Training programs	Testing programs
Rank correlation*	0.9714	0.8286
P-value**	0.00694	0.0125



Results: SDC Coverage



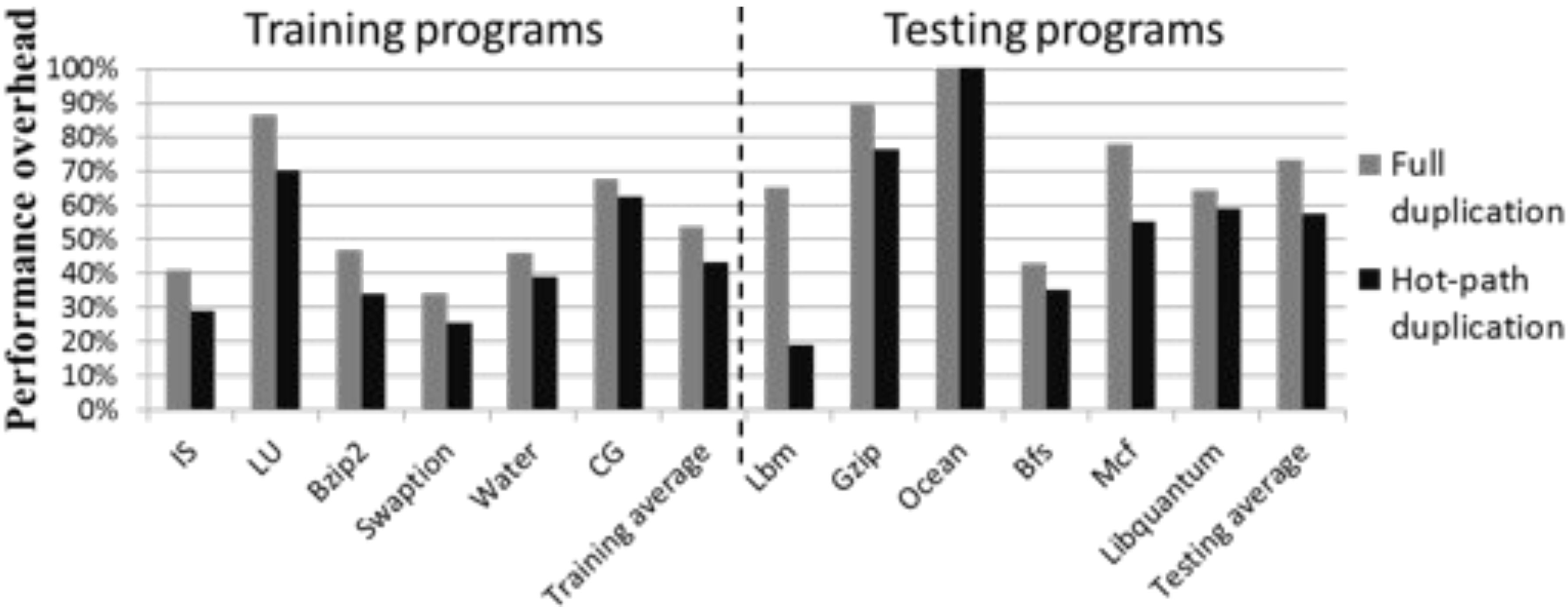
Training programs:

Overhead	Coverage
10%	44.8%
20%	78.6%
30%	86.8%

Testing programs:

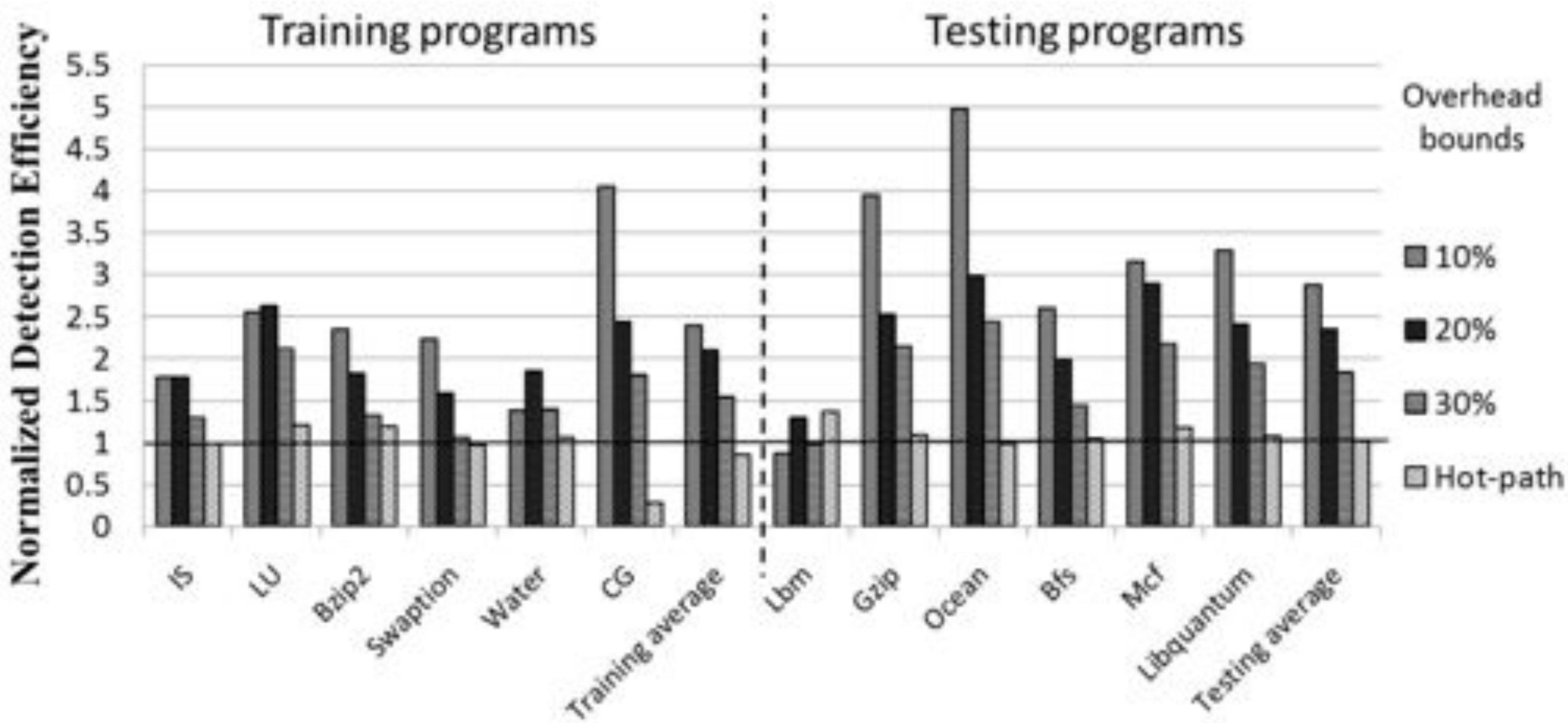
Overhead	Coverage
10%	39%
20%	63.7%
30%	74.9%

Results: Full Duplication Overheads



Full duplication and hot-path duplication (top 10% of paths) have high overheads. For full duplication it ranges from 53.7% to 73.6%, for hot-path duplication it ranges from 43.5 to 57.6%.

Results: Detection Efficiency



Normalized Detection Efficiency	10% overhead	20% overhead	30% overhead
Training programs	2.38	2.09	1.54
Testing programs	2.87	2.34	1.84

- Motivation and Goal
- Approach
- Evaluation and Results
- **Conclusion**

Conclusion and Future Work

- Configurable protection techniques for SDC failures are required as transient fault rates increase
- We find heuristics to estimate SDC proneness for program variables **based on static and dynamic features**
- SDCTune model to guide configurable SDC protection
  - Accurate at predicting relative SDC rates of applications
  - Much better detection efficiency compared to full duplication
- **Future work**
  - Improving the model's accuracy using auto-tuning
  - Using symptom based detectors for protection

http://blogs.ubc.ca/karthik/